Objectiu i plantejament de la segona part: Descobrir patrons i problemes de connectivitat

La nostra proposta és representar gràficament la qualitat de la connexió per APs mitjançant la creació de mapes de calor. I posteriorment, veure quin factor, com ara el nombre de dispositius connectats, la banda de freqüència en ús, etc., influeix en la degradació de la qualitat.

Aquesta propietat es mesura principalment a partir de la intensitat de camp/potència de senyal (dBm) trobades a les dades proveïdes de Clients. Es considera confiable a partir d'un valor de -60dBm cap amunt, mentre que un valor per sota d'aquest és considerat més dèbil i pobre a mesura que va decreixent. En l'informe anterior hem visualitzat la majoria de banda de 5GHz sobre la de 2.4GHz pel fet que s'evita saturació i interferències.

Com a alternativa, també es pot estudiar la qualitat de connexió a partir del Health Score present en les dades proveïdes de Clients. El Health Score és una mètrica que adopta un valor del 0 al 100 calculat per la pèrdua de bits basant-se en el rendiment, la connectivitat, la cobertura i el nivell d'interferència d'un AP específic, essent així 100 el màxim valor, el qual representa una bona connexió i 0, una dolenta. Intentarem veure la correlació entre les dues característiques, mitjançant altres heatmaps.

Per tal de fer això, hem processat les dades de Client.json en ordre de filtrar l'hora, el dia i el dia de la setmana a la qual es connecta l'usuari, el Health Score, la potència de senyal rebuda, i l'AP al que està connectat. Per l'altra banda, processem les dades de AP.json en ordre de filtrar la direcció IP de l'AP, el nombre d'usuaris connectats, la ubicació en format de Latitud/Longitud i l'última que està activa.

Així doncs, a més de permetre'ns ubicar cada AP individualment (assenyalat per uns quadrats translúcids de color vermell), obtenim una gràfica que inclou la qualitat de connexió de les més de mil APs que hi ha distribuïdes pel campus en àrees de color (verd per a millor connexió i roig per a pitjor) en intervals d'una hora – de manera que el jurat pot controlar la visualització mitjançant un slide arrossegable del temps. Les àrees pintades les quals estan condicionades per la mitjana de la potència de senyal segons els usuaris connectats.

$$P_m = (\Sigma P_i)/n$$

on P_m és la potència de senyal mitjana, P_i són les potències de senyals puntuals i n és el nombre de clients

$$HS_m = (\Sigma HS_i)/n\ *$$

on HS_m és el Health Score mitjà, HS_i són els Health Scores puntuals i n és el nombre de clients

Observacions:

Això implica que cada AP presentarà una circumferència del mateix diàmetre d'un únic color. D'aquesta manera contemplem el senyal d'una AP com a homogènia i ignorem les possibles interferències com podria ser el tipus i processador dels dispositius connectats o l'obstaculització per parets i objectes sòlids, entre d'altres.

L'estratègia que hem seguit ha consistit a veure i comprovar que primer és funcional per a 5 arxius i després intentar escalar el codi i els programes a macroescala. Per dur a terme tot aquest procés, ens hem fet més familiars en l'ús d'LLMs com a eina de suport, hem reforçat els coneixements en plataformes ja conegudes com ara *Python* i a més a més, hem experimentat amb nou material com ara *Folium*.

Resultats:

Amb les dades introduïdes al seu respectiu codi, obtenim com a resultat tres heatmaps diferents, que aporten tres perspectives diferents pel que fa a la valoració de la qualitat de connexió com a concepte general.

El primer de tots tracta la intensitat de camp que reben els clients connectats a una AP específica. Com ja hem esmentat abans, cada dispositiu de Wi-Fi presentarà un únic cercle d'un sol color, el qual té com a llindar el valor de -60dBm (és a dir, si els clients connectats a l'AP reben com a mitjana una potència de -60dBm, el punt serà groc. A mesura que el valor decreix, el color es va ataronjant, arribant a colors vermells si es desvia massa per sota. Anàlogament, el punt adoptarà colors més verds si el senyal que reben els clients es troba per dalt d'aquest límit). Comparant disjuntament, el diàmetre d'aquestes circumferències es mantenen igual per a cada AP.

El segon heatmap realitzat es basa en el Health Score dels clients connectats. Això ens permet diferenciar l'obtenció de senyal d'un usuari amb autèntica connexió i qualitat d'aquesta. Dotada de la mateixa dinàmica que abans, el Health Score mitjà d'una AP és representat també amb circumferències de colors de la mateixa mida per a cada, verd per a una major puntuació – 100 i vermell per a valors baixos — 0.

L'últim mapa il·lustra el nombre de clients connectats per AP, el qual ens ajuda a detectar les zones més concorregudes i possibles falles de senyal a causa de la massificació de connexió. Hem volgut visualitzar l'augment d'usuaris connectats amb l'expansió del diàmetre de la circumferència assignada per dispositiu, diàmetre més gran equival a més gent connectada i simultàniament, diàmetre més petit, menys gent connectada. Així doncs, aquest últim no presenta diferència de colors, però, en canvi, en modifiquem la grandària dels punts.

1. Filtramos por los parámetros que nos interesan

In [ ]:
```
"""
Utility script to build a lightweight JSON snapshot with the
fields needed for the ROOKIE analysis walkthrough.
```

```
It collects data from the AP and Client datasets, keeps only the
required columns, and stores them in a single JSON file:

{
    "aps": [{"timestamp": "...", "client_count": ...}, ...],
    "clients": [
        {"timestamp": "...", "hour": 12, "day_of_week": "Monday", "date":
        ...
    ]
}
"""
from __future__ import annotations

import argparse
import json
from datetime import datetime, timezone
from pathlib import Path
from typing import Iterable, Iterator, List, Optional, Tuple, Dict, Any


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Filter APS and Client datasets into a lightweight JS
    )
    repo_root = Path(__file__).resolve().parents[1]
    parser.add_argument(
        "--aps-dir",
        type=Path,
        default=repo_root / "anonymized_data" / "aps",
        help="Directory containing AP snapshot JSON files.",
    )
    parser.add_argument(
        "--clients-dir",
        type=Path,
        default=repo_root / "anonymized_data" / "clients",
        help="Directory containing client snapshot JSON files.",
    )
    parser.add_argument(
        "--output",
        type=Path,
        default=repo_root / "data" / "rookie_filtered_dataset.json",
        help="Output JSON file path.",
    )
    parser.add_argument(
        "--max-aps-files",
        type=int,
        default=None,
        help="Optional limit for AP files (useful for quick tests).",
    )
    parser.add_argument(
        "--max-client-files",
        type=int,
        default=None,
        help="Optional limit for client files.",
    )
    parser.add_argument(
        "--aps-output",
        type=Path,
        default=repo_root / "data" / "rookie_filtered_aps.json",
        help="Path for the AP-only JSON output.",
```

```
    )
    parser.add_argument(
        "--clients-output",
        type=Path,
        default=repo_root / "data" / "rookie_filtered_clients.json",
        help="Path for the client-only JSON output.",
    )
    parser.add_argument(
        "--skip-combined",
        action="store_true",
        help="Skip writing the combined JSON payload.",
    )
    parser.add_argument(
        "--aps-geojson",
        type=Path,
        default=repo_root.parent / "geolocation_package" / "data" / "aps_
        help="GeoJSON file providing AP location metadata.",
    )
    return parser.parse_args()


def iter_json_files(directory: Path, max_files: Optional[int] = None) ->
    files: List[Path] = sorted(directory.glob("*.json"))
    if max_files is not None:
        files = files[:max_files]
    for file in files:
        if not file.is_file():
            continue
        yield file


def iter_json_records(files: Iterable[Path]) -> Iterator[dict]:
    for file in files:
        with file.open("r", encoding="utf-8") as handle:
            try:
                data = json.load(handle)
            except json.JSONDecodeError as exc:
                raise ValueError(f"Invalid JSON in {file}: {exc}") from e
        if isinstance(data, list):
            for record in data:
                if isinstance(record, dict):
                    yield record


def load_geo_index(geojson_path: Path) -> Dict[str, Dict[str, Any]]:
    if not geojson_path.exists():
        return {}
    with geojson_path.open("r", encoding="utf-8") as handle:
        payload = json.load(handle)
    features = payload.get("features", [])
    index: Dict[str, Dict[str, Any]] = {}
    for feature in features:
        props = feature.get("properties", {})
        ap_name = props.get("USER_NOM_A")
        if not ap_name:
            continue
        index[ap_name] = {
            "space": props.get("USER_Espai"),
            "building_code": props.get("Nom_Edific"),
            "building_name": props.get("USER_EDIFI"),
```

```
            "floor": props.get("Num_Planta"),
            "short_ref": props.get("Ref_Curta"),
            "x": props.get("X"),
            "y": props.get("Y"),
        }
    return index


def build_aps_slice(
    directory: Path, max_files: Optional[int], geo_index: Dict[str, Dict[
) -> Tuple[List[dict], int]:
    files = list(iter_json_files(directory, max_files))
    results: List[dict] = []
    for record in iter_json_records(files):
        last_modified = record.get("last_modified")
        client_count = record.get("client_count")
        if last_modified is None:
            continue
        try:
            ts = datetime.fromtimestamp(float(last_modified), tz=timezone
        except (ValueError, TypeError):
            continue
        ts_date = ts.date().isoformat()
        ts_time = ts.time().isoformat(timespec="seconds")
        ap_name = record.get("name")
        results.append(
            {
                "name": ap_name,
                "serial": record.get("serial"),
                "timestamp": ts.isoformat(),
                "date": ts_date,
                "time": ts_time,
                "client_count": client_count,
                "location": geo_index.get(ap_name),
            }
        )
    return results, len(files)


def build_clients_slice(directory: Path, max_files: Optional[int]) -> Tup
    files = list(iter_json_files(directory, max_files))
    results: List[dict] = []
    for record in iter_json_records(files):
        last_connection = record.get("last_connection_time")
        if last_connection is None:
            continue
        try:
            # Convert from milliseconds to seconds.
            ts = datetime.fromtimestamp(float(last_connection) / 1000, tz
        except (ValueError, TypeError):
            continue
        rounded_hour = ts.hour + (1 if ts.minute >= 30 else 0)
        rounded_hour = rounded_hour % 24
        results.append(
            {
                "timestamp": ts.isoformat(),
                "hour": rounded_hour,
                "day_of_week": ts.strftime("%A"),
                "date": ts.date().isoformat(),
                "dia": ts.day,
```

```
                "health": record.get("health"),
                "signal_db": record.get("signal_db"),
                "associated_device_name": record.get("associated_device_n
            }
        )
    return results, len(files)


def main() -> None:
    args = parse_args()
    geo_index = load_geo_index(args.aps_geojson)
    aps_slice, aps_files_count = build_aps_slice(
        args.aps_dir, args.max_aps_files, geo_index
    )
    clients_slice, client_files_count = build_clients_slice(
        args.clients_dir, args.max_client_files
    )

    outputs_written = []

    if not args.skip_combined and args.output:
        output_path: Path = args.output
        output_path.parent.mkdir(parents=True, exist_ok=True)
        payload = {
            "aps": aps_slice,
            "clients": clients_slice,
            "meta": {
                "aps_files": aps_files_count,
                "client_files": client_files_count,
            },
        }
        with output_path.open("w", encoding="utf-8") as handle:
            json.dump(payload, handle, ensure_ascii=True, indent=2)
        outputs_written.append(
            f"Combined JSON â†’ {output_path} (APS {len(aps_slice)}, Clie
        )

    if args.aps_output:
        aps_path: Path = args.aps_output
        aps_path.parent.mkdir(parents=True, exist_ok=True)
        with aps_path.open("w", encoding="utf-8") as handle:
            json.dump(aps_slice, handle, ensure_ascii=True, indent=2)
        outputs_written.append(f"AP slice â†’ {aps_path} ({len(aps_slice)

    if args.clients_output:
        clients_path: Path = args.clients_output
        clients_path.parent.mkdir(parents=True, exist_ok=True)
        with clients_path.open("w", encoding="utf-8") as handle:
            json.dump(clients_slice, handle, ensure_ascii=True, indent=2)
        outputs_written.append(
            f"Client slice â†’ {clients_path} ({len(clients_slice)} regis
        )

    print("âœ… JSON generado:")
    for line in outputs_written:
        print(f"   â€¢ {line}")


if __name__ == "__main__":
    main()
```

2. Construimos las gráficas y las dejamos en formato html

In [1]:
```
import pandas as pd
import folium
from folium.plugins import TimestampedGeoJson
from pyproj import Transformer
import json
import branca # Necesario para las escalas de color
import numpy as np # Necesario para comprobar NaNs

# --- Constantes ---
FILE_APS = 'rookie_filtered_aps.json'
FILE_CLIENTS = 'rookie_filtered_clients.json'

# Archivos de salida
OUTPUT_MAP_HEALTH = 'mapa_health_dinamico.html'
OUTPUT_MAP_SIGNAL = 'mapa_signal_dinamico.html'
OUTPUT_MAP_CLIENTS = 'mapa_clientes_dinamico.html'

# --- Función de Escala (para el radio) ---
def linear_scale(value, in_min, in_max, out_min, out_max):
    """
    Mapea un valor de un rango a otro (interpolación lineal).
    Usado para calcular el radio del círculo.
    """
    if in_min == in_max:
        return (out_min + out_max) / 2
    clamped_value = max(in_min, min(value, in_max))
    in_range = in_max - in_min
    out_range = out_max - out_min
    scaled_value = (clamped_value - in_min) / in_range
    return out_min + (scaled_value * out_range)

# Definimos el conversor de coordenadas.
try:
    transformer = Transformer.from_crs("epsg:25831", "epsg:4326")
except ImportError:
    print("Error: La librería 'pyproj' no está instalada.")
    print("Por favor, instálala ejecutando: py -m pip install pyproj bran
    exit()

# --- ¡MODIFICADO! Coordenadas de inicio ---
# Coordenadas de AP-VET71
start_x, start_y = 424638.107049, 4595093.80301
start_lat, start_lon = transformer.transform(start_x, start_y)
map_center_coords = [start_lat, start_lon]
# ------------------------------------

print("Script iniciado...")

# --- 1. Cargar y Procesar Datos de Clientes ---
print(f"Cargando y procesando clientes desde {FILE_CLIENTS}...")
try:
    df_clients = pd.read_json(FILE_CLIENTS)

    df_clients['health'] = pd.to_numeric(df_clients['health'], errors='co
    df_clients['signal_db'] = pd.to_numeric(df_clients['signal_db'], erro
    df_clients = df_clients.dropna(subset=['health', 'signal_db', 'associ
```

```
    df_metrics = df_clients.groupby(['date', 'hour', 'associated_device_n
        avg_health=('health', 'mean'),
        avg_signal_db=('signal_db', 'mean'),
        num_clients_metricos=('health', 'size')
    ).reset_index()

    df_metrics.rename(columns={'associated_device_name': 'name'}, inplace
    print(f"Métricas de clientes calculadas (ej: {len(df_metrics)} regist

except FileNotFoundError:
    print(f"Error: No se encontró el archivo {FILE_CLIENTS}")
    exit()
except Exception as e:
    print(f"Error procesando {FILE_CLIENTS}: {e}")
    exit()

# --- 2. Cargar y Procesar Ubicaciones de APs ---
print(f"Cargando y procesando APs desde {FILE_APS}...")
try:
    with open(FILE_APS, 'r', encoding='utf-8') as f:
        data_aps = json.load(f)
    df_aps = pd.DataFrame(data_aps)

    df_aps = df_aps.dropna(subset=['location'])
    df_ap_locations = df_aps.drop_duplicates(subset=['name'], keep='last'

    def convert_coordinates(row):
        try:
            x = row['location']['x']
            y = row['location']['y']
            lat, lon = transformer.transform(x, y)
            return pd.Series([lat, lon, row['location'].get('building_nam
        except Exception:
            return pd.Series([None, None, None])

    print("Convirtiendo coordenadas UTM a Lat/Lon (esto puede tardar un m
    df_ap_locations[['lat', 'lon', 'building_name']] = df_ap_locations.ap
    df_ap_locations = df_ap_locations.dropna(subset=['lat', 'lon'])
    df_ap_locations = df_ap_locations[['name', 'lat', 'lon', 'building_na
    print(f"Ubicaciones únicas de APs procesadas (total: {len(df_ap_locat

except FileNotFoundError:
    print(f"Error: No se encontró el archivo {FILE_APS}")
    exit()
except Exception as e:
    print(f"Error procesando {FILE_APS}: {e}")
    exit()

# --- 3. Unir Métricas y Ubicaciones ---
print("Uniendo métricas de clientes con ubicaciones de APs...")
df_master = pd.merge(df_metrics, df_ap_locations, on='name', how='inner')

if df_master.empty:
    print("Error: No se ha podido encontrar datos comunes entre clientes
    exit()

# Creamos el timestamp string en el dataframe maestro
df_master['hour_str'] = df_master['hour'].astype(str).str.zfill(2)
df_master['timestamp_str'] = df_master['date'].dt.strftime('%Y-%m-%d') +
```

```
# --- 4. Preparar Datos para TimestampedGeoJson (¡MODIFICADO!) ---
print("Creando 'scaffolding' de tiempo/AP para evitar 'stacking'...")

# Obtenemos todos los APs únicos y todos los tiempos únicos
all_aps_data = df_ap_locations[['name', 'lat', 'lon', 'building_name']]
all_times = df_master['timestamp_str'].unique()
all_times.sort() # Nos aseguramos de que el tiempo esté ordenado

# 1. Crear el "andamio" (scaffolding) con todas las combinaciones posible
df_scaffold_index = pd.MultiIndex.from_product([all_aps_data['name'].uniq
df_scaffold = pd.DataFrame(index=df_scaffold_index).reset_index()

# 2. Unir el andamio con los datos de AP (para tener lat/lon siempre)
df_master_full = pd.merge(df_scaffold, all_aps_data, on='name', how='left

# 3. Unir con los datos de métricas (esto creará 'NaN' donde no haya dato
df_master_full = pd.merge(
    df_master_full,
    df_master,
    on=['name', 'timestamp_str', 'lat', 'lon', 'building_name'],
    how='left'
)

print(f"Formateando datos GeoJSON para los mapas dinámicos (Total feature

# --- Definir escalas de color y tamaño ---
cmap_bueno_es_verde = branca.colormap.LinearColormap(['red', 'yellow', 'g
max_clients_global = df_master['num_clients_metricos'].max()
if pd.isna(max_clients_global) or max_clients_global == 0: max_clients_gl
cmap_mucho_es_rojo = branca.colormap.LinearColormap(['green', 'yellow', '

# Estilo INVISIBLE para APs sin datos
style_invisible = {
    'color': '#000000', 'fillColor': '#000000',
    'opacity': 0.0, 'fillOpacity': 0.0, 'weight': 0, 'radius': 0
}

# --- Función para crear las "features" de GeoJSON (corregida) ---
def create_feature(row, timestamp, iconstyle, popup):
    """Crea una única feature de GeoJSON para un punto en el tiempo."""
    return {
        'type': 'Feature',
        'geometry': {
            'type': 'Point',
            'coordinates': [row['lon'], row['lat']]
        },
        'properties': {
            'time': timestamp,
            'icon': 'circle',
            'iconstyle': iconstyle,
            'popup': popup
        }
    }

features_health = []
features_signal = []
features_clients = []
```

```
# Iteramos sobre el dataframe COMPLETO (df_master_full)
for _, row in df_master_full.iterrows():
    ts = row['timestamp_str']

    # Comprobamos si hay datos para esta hora/AP
    is_active = not pd.isna(row['avg_health'])

    if is_active:
        # --- Si está ACTIVO, creamos estilos VISIBLES ---
        popup_html = (f"<b>AP:</b> {row['name']}<br>"
                      f"<b>Edificio:</b> {row['building_name']}<br>"
                      f"<b>Hora:</b> {ts}<br>"
                      f"<b>Health:</b> {row['avg_health']:.1f}<br>"
                      f"<b>Señal:</b> {row['avg_signal_db']:.1f} dBm<br>"
                      f"<b>Clientes:</b> {row['num_clients_metricos']}")

        # 1. Estilo Health
        health_color = cmap_bueno_es_verde(row['avg_health'])
        style_health = {
            'color': health_color, 'fillColor': health_color,
            'opacity': 0.8, 'fillOpacity': 0.6, 'weight': 1, 'radius': 15
        }

        # 2. Estilo Signal
        signal_weight = (100 + row['avg_signal_db'])
        signal_color = cmap_bueno_es_verde(signal_weight)
        style_signal = {
            'color': signal_color, 'fillColor': signal_color,
            'opacity': 0.8, 'fillOpacity': 0.6, 'weight': 1, 'radius': 15
        }

        # 3. Estilo Clientes
        client_radius = linear_scale(row['num_clients_metricos'], 0, max_
        client_color = cmap_mucho_es_rojo(row['num_clients_metricos'])
        style_clients = {
            'color': client_color, 'fillOpacity': 0.0, 'opacity': 0.7,
            'weight': 3, 'radius': client_radius
        }

    else:
        # --- Si está INACTIVO, creamos estilos INVISIBLES ---
        popup_html = f"<b>AP:</b> {row['name']}<br><b>Hora:</b> {ts}<br>S

        style_health = style_invisible
        style_signal = style_invisible
        style_clients = style_invisible

    # Añadimos la feature (visible o invisible)
    features_health.append(create_feature(row, ts, style_health, popup_ht
    features_signal.append(create_feature(row, ts, style_signal, popup_ht
    features_clients.append(create_feature(row, ts, style_clients, popup_

print(f"Datos GeoJSON preparados.")

# --- 5. Función para crear y guardar los mapas (¡MODIFICADA!) ---
def create_dynamic_bubble_map(features_list, ap_locations, output_filenam
    print(f"Creando mapa: {output_filename}...")

    # --- ¡CAMBIO! Centramos en las coordenadas dadas con zoom 16 ---
    m = folium.Map(location=map_center_coords, zoom_start=16)
```

```
    # Capa 1: Marcadores de APs
    fg_aps = folium.FeatureGroup(name='Mostrar Ubicación de APs')
    offset_lat = 0.00003
    offset_lon = 0.00004
    for _, ap in ap_locations.iterrows():
        bounds_rect = [
            [ap['lat'] - offset_lat, ap['lon'] - offset_lon],
            [ap['lat'] + offset_lat, ap['lon'] + offset_lon]
        ]
        folium.Rectangle(
            bounds=bounds_rect,
            color="#e63946", fill=True, fill_color="#e63946", fill_opacit
            popup=f"<b>AP:</b> {ap['name']}<br><b>Edificio:</b> {ap['buil
        ).add_to(fg_aps)
    fg_aps.add_to(m)

    # Capa 2: Círculos Dinámicos
    TimestampedGeoJson(
        {'type': 'FeatureCollection', 'features': features_list},
        period='PT1H',
        duration='PT1H', # <-- ¡ARREGLO PARA "STACKING"! (Cada círculo du
        add_last_point=False, # <-- No dejar el último punto
        auto_play=False,
        loop=False,
        max_speed=100, # <-- ¡VELOCIDAD AUMENTADA!
        loop_button=True,
        date_options='YYYY-MM-DD HH:mm',
        time_slider_drag_update=True,
    ).add_to(m)

    # Título
    title_html = f'''
                 <div style="position: fixed; top: 10px; left: 50px; z-in
                             font-size: 24px; font-weight: bold; color: #
                             background-color: rgba(255, 255, 255, 0.7);
                             padding: 5px 15px; border-radius: 5px;">
                   {map_title} (UAB)
                 </div>
                 '''
    m.get_root().html.add_child(folium.Element(title_html))

    folium.LayerControl().add_to(m)
    m.save(output_filename)
    print(f"¡Mapa guardado! -> {output_filename}")

# --- 6. Generar los TRES mapas ---
create_dynamic_bubble_map(
    features_health,
    df_ap_locations,
    OUTPUT_MAP_HEALTH,
    "Mapa Dinámico: Health (Color: 0=Rojo, 100=Verde)"
)

create_dynamic_bubble_map(
    features_signal,
    df_ap_locations,
    OUTPUT_MAP_SIGNAL,
    "Mapa Dinámico: Señal (Color: Malo=Rojo, Bueno=Verde)"
)
```

```
create_dynamic_bubble_map(
    features_clients,
    df_ap_locations,
    OUTPUT_MAP_CLIENTS,
    "Mapa Dinámico: Nº Clientes (Tamaño: Dinámico | Borde: Verde-Rojo)"
)

print("\n¡Proceso completado! Revisa los TRES archivos .html generados.")
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 2
      1 import pandas as pd
----> 2 import folium
      3 from folium.plugins import TimestampedGeoJson
      4 from pyproj import Transformer

ModuleNotFoundError: No module named 'folium'
```

3. Lo implementamos mediante dashboard

Què n'extraiem d'aquests heatmaps?

Tots tres ens aporten informació bàsica sobre les connexions com ara les hores punta i dies que freqüenten més gent. En podem extreure que, sens dubte, en dies lectius hi ha un flux major que no pas els caps de setmana gràcies a les icòniques aparicions en massa de circumferències i que són més ostentoses i visibles en horari acadèmic; les connexions són nombroses durant el matí i el migdia i es van reduint a mesura que es fa tard. A més a més, cada propietat ens aporta informació sobre possibles problemes a tenir en compte en la xarxa d'instal·lació d'APs. L'estudi en vers la potència de senyal ens permet localitzar punts-vall on no arriba bé el senyal; la investigació sobre el Health Score ens ajuda a identificar dificultats de connectivitat dels clients; i finalment el nombre de clients ens ensenya bàsicament la densitat d'usuaris per AP, el qual ens pot ajudar a redistribuir APs segons zones on hi hagi més o menys demanda per saturació.

Conclusions

Del primer heatmap concloem que la potència de senyal és bastant constant en tota la universitat, amb colors entre groc i taronja i potser algun valor atípic que torna un parell de circumferències roges o més verdes.

Per Health Score, veiem que la majoria de les connexions són compatibles i positives, un vel que cobreix el campus majoritàriament verd. Tot i que momentàniament apareixen cercles grocs, taronges o rojos.

En el cas de nombre de clients, podem veure la densitat d'usuaris per AP.

Amb els mapes obtinguts hem intentat realitzar un mapa de calor en tres dimensions amb l'objectiu de relacionar la qualitat de connexió amb factors que podrien ser

causants de la disminució d'aquesta com ara massa gent connectada al mateix AP. Així aprofitaríem la relació de dades com a mesura per descartar problemes alterns en vers altres variables. Desafortunadament, no ho hem pogut posar en pràctica, ja que els recursos necessaris per fer tal treball no eren compatibles amb les versions de les eines que posseíem.

Complicacions:

- Ordre de magnitud de dades massa grans per analitzar, emmagatzematge incompatible amb els ordinadors disponibles de l'equip.
- Existència del retorn "nul" en una coordenada en filtrar les dades (potencial error en el moment de crear els heatmaps). Més tard, analitzant els heatmaps produïts, ens vam adonar que existia un sol AP en mig de l'oceà Atlàntic, el qual no té gens de sentit pensant que estem investigant repetidors en el domini de la UAB. D'això vam extreure la conclusió que coincidia amb les coordenades (0,0), possiblement a causa de la interpretació "nul" del codi.
- Llibreries inservibles per a versions de Python més recents en l'intent de transferir informació per a fer un heatmap tridimensional.
- En un moment més avançat del projecte, hem pogut córrer la simulació dels mapes de calor adquirits amb totes les dades subministrades. Vam detectar un problema en què s'stackejaven les àrees pintades, el qual dificultava la manipulació dels heatmaps amb els processadors disponibles.

Futures ampliacions: La nostra idea és poder entrenar una intel·ligència artificial amb els heatmaps aconseguits perquè detecti automàticament zones problemàtiques mitjançant el criteri d'anàlisi seguit en el projecte. Així facilitaríem la recol·locació i l'addició d'APs en cas que es doni i també obtindríem prediccions sobre el posicionament ideal de dispositius i errors prevenibles.

In [ ]: